JQuery UI like approximate autocomplete

Ibrahim Chegrane, Djamal Belazzougui, Mathieu Raffinot

JQuery UI like approximate autocomplete

Ibrahim Chegrane
LRIA, USTHB, Algeria.
ibra.chegrane@gmail.com

Djamal Belazzougui
Dep. of Computer Science FI-00014 Univ. of Helsinki, Finland.

Mathieu Raffinot
CNRS, LIAFA, Université Paris Diderot –Paris 7, France.

Abstract

Approximate auto complete facilitates and speeds up input writing by offering a list of suggestions which complements the few characters typed by the user. In this paper we present a method based on a trie to do an efficient 1 edit error approximate auto complete in client server architecture, and we discuss different strategies to improve the efficiency of auto completion, depending on different scenarios. We also propose a method that reduces the number of outgoing transition tested in each node - especially in the first levels - of the trie. It uses a hash based index to generate candidate characters to be tested at each branch.

We present a library (named `appacolib`*), in fact a set of different language libraries, to be of use either on the server or the client browser or both to rapidly answer approximate requests on an* `UTF-8` *dictionary.*

I. Introduction

Text fields are used to enter data to the computer, and they have been improved across the years. Today, they are equipped with many features to help user, and among the most interesting one we find auto completion (also called auto suggestion), which is a technique that facilitates and speeds up writing, by offering a list of words or phrases (suggestions) complementing the few characters typed into the text input, in a very short time (usually few milliseconds). The usual, but not mandatory, way those words are chosen is that they accept the few characters typed as prefix. Then, considering this list, the user can either choose one of the words or continue typing characters, which brings up new lists of words. Auto completion is very useful and popular in several domains and systems. In the web, this feature is provided by many web browsers to complement urls (e.g. Mozilla Firefox, Google Chrome). On desktop computers this feature is integrated in many applications, "Tab" key when using a command line interpreter (bash shell in UNIX).

Considering auto-suggestion programming libraries, one of the most used auto-suggest library for allowing programmer to add such a feature on they input field is JQuery UI auto-complete (see `http://api.jqueryui.com/autocomplete/`).

However, and this is the main motivation of this work, the input string (the few characters) being typed may contain errors. It's maybe a keystroke error, especially when typing fast, or the user don't know the correct spelling (person name, product name...etc.). JQuery UI autocomplete and other auto-complete libraries we are aware of do not allow typing errors. This limits the suggestions of the auto-complete system to exact prefix matches, which can thus be incomplete or misleading.

To solve this problem we have to tolerate some number of errors in the prefix typed, and in the list of suggestions to obtain a list exact and approximate completions. The most commonly used metric to determine the difference between two strings x and y is called "edit distance" [12] that is defined as follow: the minimum number of operations needed on string x to get string y using three basic operations: deletion, insertion, and substitution.

A direct consequence of tolerating errors is that the list of suggestions might be very large; to reduce the number of results we thus propose a new auto-complete JQuery like library (named `appacolib`) which (a) limit the number of possible error to at most 1 and (b) report k most highly ranked suggestions in decreasing order of their ranks. We call this the top-k suggestions; the parameter k is given by the user. The `appacolib` library, in fact a set of different language libraries, can be used either on the server (C/C++) or the client (JavaScript)

or both to rapidly answer approximate requests on an UTF-8 dictionary. When dealing with real world data, practical problems appear that are usually not taken into account in theoretical work. In particular, managing UTF-8 files efficiently in time and memory space is not that technically trivial. Our libraries permit to deal with UTF-8 dictionaries, whatever the underlying language.

Related work The concept of auto completion is not new, in 1990 J. Darragh et al. have developed *The Reactive Keyboard* [5] predicting what the user is going to type next. There are various approaches that work on different auto completion strategies [3] depending on whether we do exact and approximate auto complete, complete to an word, a substring of a phrase. Bast and Weber [1] propose an indexing method for word completion based on a document corpus. There are lot of studies on extending auto-completion to tolerate errors [3, 16, 10] using edit distance constraints, and different data structures (e.g q-gram and trie based completion). Since the list of suggestions can be very large only the the top-k most highly ranked suggestions are reported [9, 13] using a ranked trie in which a score is stored in leaves. Phrase prediction or sentence completion is more complicated than simple word completion, the completion is based on a given corpus [14, 8]. On client/server side Google present on-line searches suggests query auto completion strings (terms and/or phrases), based on database access system [15, 7].

II. Preliminaries

A *trie* or a *prefix tree* is an ordered data structure used to index all dictionary words in order to search and get a response in time linear in the length the query word m. The edges are labeled by string characters; nodes represent a common prefix of a set of strings; all the outgoing transitions from one node have the same prefix; each word can be found by traversing the trie from the root to the leaf by following the characters on the edges.

The trie can be compacted, and thus called a *compact trie* (also sometimes a PATRICIA tree) by merging any node that is an only child with its parent and concatenating the labels of the transitions which become multi-characters transitions. An important detail is that then each transitions is in fact labeled with a factor of one of the words in the dictionary. Thus, to reduce the memory required but still remain efficient, we code a transition by its first character, a pointer into the dictionary to the beginning of an occurrence of the corresponding factor, and the length of the transition. This implies that in this approach we keep both the dictionary and the trie index in main memory.

To efficiently build our compact trie, we also use a longest common prefix array (LCP array) [11] which is a table that stores the lengths of the longest common prefixes between each pair of consecutive words of the dictionary.

Most of the efficient approximate prefix searching algorithms first build a type of index on the dictionary and then use this index to perform each query efficiently. We build an index also, that can be compared to a ranked compact trie of all dictionary words. However, to keep this structure as small as possible, uft8 characters and integers (pointers are also coded as integers) are byte-coded [6].

Trie construction algorithm main lines: 1. order the dictionary in lexicographic order; 2. build the LCP array of all dictionary words; 3. build the compacted Trie based on LCP array to speed up the construction; 4. add score to the compacted trie : each word is associated with a static score, in the compacted Trie construction when we reach the leaf we store the word score. After the insertion of all words in the trie, to support efficient top-k completion, for each intermediate node we recursively keep the maximum score among its children.

III. Search Method

We first begin by explaining the algorithm to search a result and propose a suggestion list. In our method the trie is used to answer queries in an approximate way, using a ranked traversal

of some promising nodes of the trie. Let w be a query word where $|w| = m$, and k is the number of requested completions.

Find valid locus Given a word w, we name *locus* the position in the compacted trie on the node or in the middle of the edge that represents the position such that we can't go forward in the trie in the exact search for w. Because either we get to depth $|w|$ or we get an error before we find the whole word w.

We name *locus node* the node that represents a locus. If the locus ends in a node, this node is the locus node; otherwise, the locus ends in the middle of a edge and we consider the node at the end of the edge as the locus node.

All_valid_nodes algorithm:
Input: query prefix w
Output: a list of valid nodes

a. Find the locus node nd of w.

b. If the locus node nd is at depth at least $|w|$, we find an exact solution, add this node to the list of valid node.

c. For each node on the lead path to the locus node found at (a):

- Take the node as a locus node
- Do an edit distance operation on all outgoing transitions (except for the one that is on the lead path to the locus node founded at (a))
- Continue an exact search of the remaining suffix of the query in the subtree of this new locus node.
- If an approximate match of w is found we add the locus node of this solution as a valid node to the list of solution.

Get list of suggestion results Algorithm to compute all top-k suggestion lists once the set of valid nodes locus have been computed.

List_of_suggestions algorithm
Input: list of valid nodes
Output: list of top-k completion.

a. There is an exact solution: we take all outgoing transition nodes with their score (of an exact valid node) and add them to the priority queue, iteratively, we get the node with the largest score which is on the top of the priority queue. If this node represents a leaf add the word corresponding into the list of completions. Otherwise, insert its children in the priority queue. Do this operation until we will have k words in the list of suggestion or the priority queue is empty.

b. If the priority queue is empty before we have k words in the list of suggestions or we haven't an exact solution: we add all the remaining valid nodes that represent an approximate solution with their score into the priority queue, and we do the same thing until we complete the k words into the list of suggestions or until the priority queue is empty again.

Auto completion and user typing. In general, the user types his query character after character and can modify the string using any of combination of keystrokes; the user can append a character, insert or delete a character either at the end or anywhere in the string. Thus, not to recompute all locus nodes from scratch each time the user types a new key, we remember the sets of locus positions after each new key and try to save some computation by starting the search from them when possible.

I. On the first few key stroke of the query: we perform **All_valid_nodes** and **List_of_suggestions** algorithms.

II. When characters are appended at the end: At each new keystroke we don't begin the search from the root of the trie, instead, the locus position in the last exact search will be considered as the new root and we continue to search from this position and we do exactly the same step in **All_valid_nodes** and **List_of_suggestions**.

III. Modification (deletion, insert in the middle...etc.):

We save all the nodes on the lead way in a table (along with the length of the longest path in the trie); we store each node in a bucket that corresponds to the depth of the character in this position. If there is a common prefix between the modified query prefix and the previous one, we find the node corresponding to the same prefix from the table stored before, we go directly to the bucket in position (depth of common prefix), this node will be considered as a root and we continue the search and do same step of algorithm 2 and 3. In general, if there is a common prefix between the two query words, start the search from the node in this level (depth of common prefix).

IV. Reduce number of outgoing branch tested

In order to reduce the number of outgoing branches, we experimented on the server side version with using a hash based index to generate candidate characters to be tested at each branch. We store substitution lists as defined in [4, 2], but only on prefixes of bounded length. For every length we store a dictionary that stores candidate characters for every positions. A substitution dictionary for length d stores lists of characters c associated with substitution patterns $p?q$, such that pcq is a string of length d which is prefix of some string in the dictionary. The dictionary is approximate in the sense that, given a substitution pattern $p?q$, it could return a superset of the set of characters c that, when substituted to the ?, generate a prefix of a string in the dictionary. In our case, we will have a parameter D (for example $D = 6$), such that we will store substitution lists for all prefixes of length $d \leq D$. We implemented two strategies. Given a query string $p[1..m]$ with $m \leq D$, in the first strategy, we generate all candidate prefixes by querying the substitution list dictionary m times for patterns $p[1..i]?[i+1..m]$, for $i \in [1..m]$. In the second strategy, we query the candidate dictionary while traversing the trie top-down. More precisely supposing for a traversed node at depth d, we query the substitution list dictionary for the substitution pattern $p[1..d-1]?P[d+1..m]$ to get a list of characters. We then do the intersection of this list with the list of characters that label the children of the node and only continue traversing the nodes labeled by the characters in the intersection. We have tested both strategies and the second one seems to give much better results. The space is only increased by very small amount.

V. Client/server strategy

There are two extreme strategies for autocompletion: either the program runs on the server (for instance, apache + daemon) and the client only sends queries and manages results; or the program runs on the client (using JavaScript), and the client only sends the final choice. In between these two extremes, many scenarios might be considered, depending on the following important parameters: (a) Dictionary size; (b) Server load; (c) Connection speed; (d) Data volume; (e) Number of concurrent connections on the server; (f) Computing power of the client (the power of server is supposed to be $\gg$ more than this power, but shared); (g) Static vs dynamic rank; (e) Local vs Global dynamically ranked dictionary; (f) Update delay.

Our libraries available in `https://github.com/AppacoLib/api.appacoLib` enable us to execute our algorithms either on the server or on the client side, depending on the best scenario one is searching for. Our core libraries are written in C/C++ on the server side and in JavaScript on the client side. We explain below its two main uses.

On server side, typically if we have a very big dictionary (more than 1M entries), we can use the library written in C/C++ with a FASTCGI module to provide a list of suggestions in JSON format, and query and send the results back through AJAX calls and then display the list using our library written in JavaScript (usually below an auto complete input). For a demonstration, see `http://appacolib.xyz/result_from_server_CGI.html`.

On client side, we have many strategies depending on the needed scenario, for instance: 1) all approximate auto-complete might be performed in the local browser on the client side. If the dictionary is not that huge (for instance less than 1M entries), we can do all operations in local: the construction of the index, the search of the query prefix, and the display of the list of suggestions. In this case we must provide a list of words to construct the index, the source of dictionary can be local or on the server and can be in different format (file, string, array). 2) display the result only in the client side: use an Ajax call to get a list from the server side. 3) download the trie already built from the server and perform the queries locally.

When displaying top-k results, also we provide an option to get other results if they are still valid nodes in the priority queue, and display them group by group. For a demonstration, see `http://appacolib.xyz`, `next` button.

VI. Test and experiment

Test have been done on two files, an English dictionary (213557 words, 2.4 Megabytes), and an extract or Wikipedia article tiles (1200000 titles, 11.5 Megabytes). We considered the two extreme scenarios cited above and performed some time tests on trie building and requests and top-k answers (setting $k = 10$) for query length varying from 2 to 6. The query strings were obtained by randomly choosing among prefixes of strings from the dictionary and introducing random errors at random positions. All the times were obtained by averaging over 1000 distinct query strings.

Server side, C/C++ Tests have been performed on an Intel core-2 duo e8400, windows 7, 3.0 GHz, 2GB RAM, GNU GCC Compiler version 4.4.1. The space occupied by the index is about 5 Megabytes for the English dataset and 29 Megabytes for the Wikipedia dataset.

For exact searches, it is interesting to note that the answer time for short queries (len = 2) are longer than long queries (len = 6), mainly because there are much more results under a fixed threshold when the query is short. However, for 1-error searches, the trie traversal time is much higher and the time difference in reporting results still exist but becomes negligible compared to the whole response time.

Query length	Exact (En)	1-error (En)	Exact (Wi)	1-error (Wi)
2	0.02	0.11	0.02	0.34
3	0.017	0.16	0.017	0.44
4	0.013	0.19	0.014	0.53
5	0.010	0.21	0.012	0.58
6	0.009	0.25	0.01	0.61

Table 1: *Query + top-k times (in milliseconds) on English dictionary (En - 213557 words) and on WikiTitles (Wi - 1200000 titles). Trie building for En is about 177 ms and 980 ms on Wi.*

Client side, JavaScript Tests have been performed on a pc Netbook (now less powerful than a SmartPhone), with a processor Intel Atom CPU N455 1,67 Ghz, 2Go RAM and 32 bits window 7 starter.

Query length	Exact (En)	1-error (En)	Exact (Wi)	1-error (Wi)
2	0.006	0.47	0.0123	1.88
3	0.009	0.59	0.0242	2.41
4	0.0114	0.65	0.0186	2.42
5	0.0125	0.62	0.0206	2.28
6	0.0143	0.58	0.0233	2.08

Table 2: *Chrome Browser. Query + top-k times (in milliseconds) on En and Wi. Trie building for En is about 2.7 seconds and 13 seconds on Wi.*

The best performance was obtained with Opera 26, followed by Google Chrome (slightly behind), then Firefox 29 and Finally Microsoft Internet Explorer. For lack of space, we presented the results only two browses: Chrome, which is a popular browser and was only slightly slower than the fastest one and Internet Explorer which was the slowest.

Substitution List, Server side We experimented with the use of substitution list dictio-

Query length	Exact (En)	1-error (En)	Exact (Wi)	1-error (Wi)
2	0.0471	1.64	0.0868	7.07
3	0.0492	1.93	0.0883	8.72
4	0.0608	2.37	0.1063	10.56
5	0.0896	3.29	0.1172	11.07
6	0.1013	3.59	0.1308	10.59

Table 3: *Internet Explorer browser. Query + top-k times (in milliseconds) on En and Wi. Trie building for En is about 30 s. and 150 s. on Wi.*

nary. We experimented only with the second strategy since it gave better results. Interestingly, one can see that, when using the substitution dictionary, the search time decreases when we increase the query length. This is due to the fact that the size of the substitution lists are shorter for long prefixes than short ones (intuitively a longer substitution pattern $p?q$ will match less prefixes than a shorter one).

Query length	No SL (En)	with SL (En)	No SL (Wi)	with SL (Wi)
2	0.11	0.11	0.34	0.37
3	0.16	0.13	0.44	0.4
4	0.19	0.11	0.53	0.32
5	0.21	0.09	0.58	0.2
6	0.25	0.06	0.61	0.13

Table 4: *Query time with and without the use of substitution lists for prefixes of lengths up to* 6. *The substitutions lists increased the indexes size by 0.7 Megabytes for English and 5 Megabytes for Wiki.*

References

[1] H. Bast and I. Weber. Type less, find more. In *Proceedings of the 29th annual international ACM SIGIR conference on Research and development in information retrieval - SIGIR '06*, page 364, New York, New York, USA, Aug. 2006. ACM Press.

[2] D. Belazzougui. Faster and space-optimal edit distance "1" dictionary. In *Combinatorial Pattern Matching, 20th Annual Symposium, CPM 2009, Lille, France, June 22-24, 2009, Proceedings*, pages 154–167, 2009.

[3] S. Chaudhuri and R. Kaushik. Extending autocompletion to tolerate errors. In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of Data*, SIGMOD '09, pages 707–718, New York, NY, USA, 2009. ACM.

[4] I. Chegrane and D. Belazzougui. Simple, compact and robust approximate string dictionary. *J. Discrete Algorithms*, 28:49–60, 2014.

[5] J. Darragh, I. Witten, and M. James. The Reactive Keyboard: a predictive typing aid. *Computer*, 23(11):41–49, Nov. 1990.

[6] P. Elias. Universal codeword sets and representations of the integers. *IEEE Transactions on Information Theory*, 21(2):194–203, Mar. 1975.

[7] K. A. Gibbs. Method and system for URL autocompletion using ranked results, 2009.

[8] K. Grabski and T. Scheffer. Sentence completion. In *Proceedings of the 27th annual international conference on Research and development in information retrieval - SIGIR '04*, page 433, New York, New York, USA, July 2004. ACM Press.

[9] B.-J. P. Hsu and G. Ottaviano. Space-efficient data structures for top-k completion. In *Proceedings of the 22Nd International Conference on World Wide Web*, WWW '13, pages 583–594, Republic and Canton of Geneva, Switzerland, 2013. International World Wide Web Conferences Steering Committee.

[10] S. Ji, G. Li, C. Li, and J. Feng. Efficient interactive fuzzy keyword search. In *Proceedings of the 18th International Conference on World Wide Web*, WWW '09, pages 371–380, New York, NY, USA, 2009. ACM.

[11] T. Kasai, G. Lee, H. Arimura, S. Arikawa, and K. Park. Linear-Time Longest-Common-Prefix Computation in Suffix Arrays and Its Applications. In *Proceedings of the 12th Annual Symposium on Combinatorial Pattern Matching*, CPM '01, pages 181–192. Springer-Verlag, 2001.

[12] V. I. Levenshtein. Binary Codes Capable of Correcting Deletions, Insertions, and Reversals. *Soviet Physics Doklady*, 10:707–710, 1966.

[13] D. Matani. An O (k log n) algorithm for prefix based ranked autocomplete. pages 1–14, 2011.

[14] A. Nandi and H. V. Jagadish. Effective phrase prediction. In *Proceedings of the 33rd International Conference on Very Large Data Bases*,

VLDB '07, pages 219–230. VLDB Endowment, 2007.

[15] R. E. Ortega, J. W. Avery, and R. Frederick. Search query autocompletion, 2003.

[16] C. Xiao, J. Qin, W. Wang, Y. Ishikawa, K. Tsuda, and K. Sadakane. Efficient error-tolerant query autocompletion. *Proc. VLDB Endow.*, 6(6):373–384, Apr. 2013.